

# Selections

Introduction to Programming and  
Computational Problem Solving:  
Accelerated Pace

CSE 11

Lecture 5

# Announcements

- Assignment 2 is due Oct 15, 11:59 PM
  - Upgrade beginning Oct 18, 12:01 AM
- Assignment 3 will be released Oct 15
  - Due Oct 22, 11:59 PM

# Selections

- Relational operators (e.g., less than, equal to)
- Logical operators (e.g., not, and, or)
- `if` statements
- `if-else` statements
- `switch` statements

# The `boolean` type and operators

- Often in a program you need to compare two values, such as whether `i` is greater than `j`
- Java provides six comparison operators (also known as relational operators) that can be used to compare two values
- The result of the comparison is a Boolean value: `true` or `false`
- For example  

```
boolean b = (1 > 2);
```

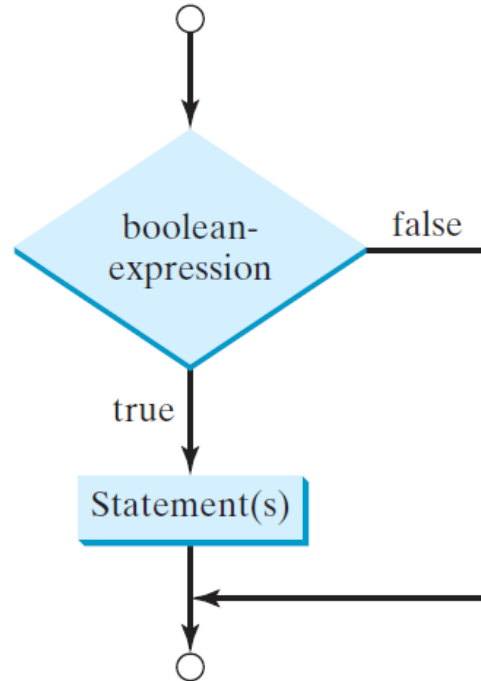
# Relational operators

Java Operator	Mathematics Symbol	Name	Example (radius is 5)	Result
<	<	less than	<code>radius &lt; 0</code>	<code>false</code>
<=	≤	less than or equal to	<code>radius &lt;= 0</code>	<code>false</code>
>	>	greater than	<code>radius &gt; 0</code>	<code>true</code>
>=	≥	greater than or equal to	<code>radius &gt;= 0</code>	<code>true</code>
==	=	equal to	<code>radius == 0</code>	<code>false</code>
!=	≠	not equal to	<code>radius != 0</code>	<code>true</code>

# if statements

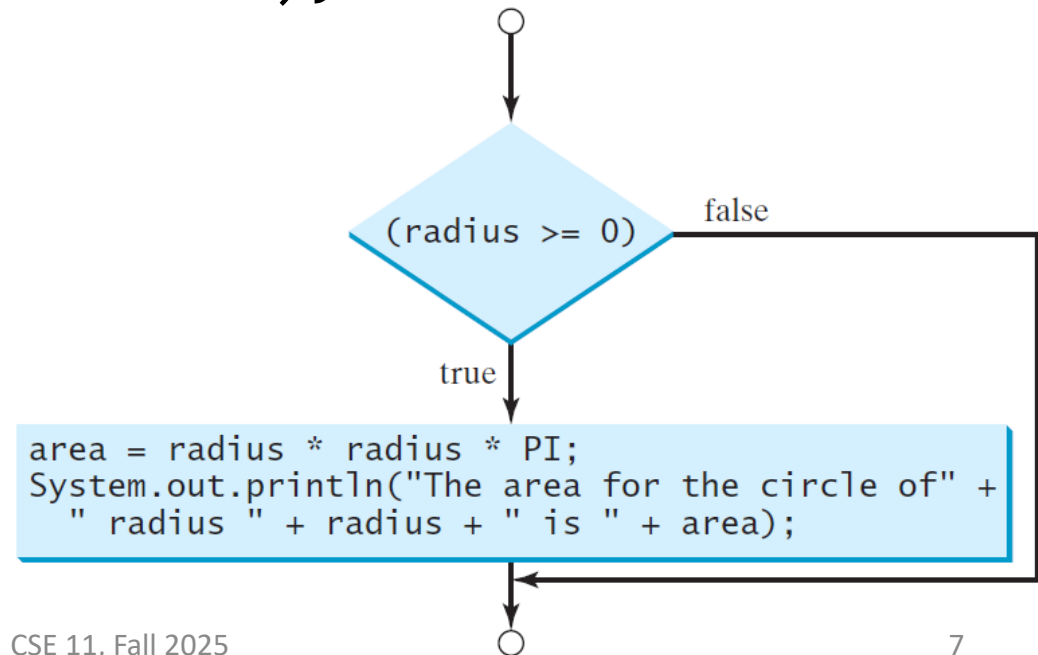
```
if (boolean-expression) {  
    statement(s);  
}
```

Braces are optional for a single statement; however, it is best practice (less error prone) to **always use braces**



# if statements

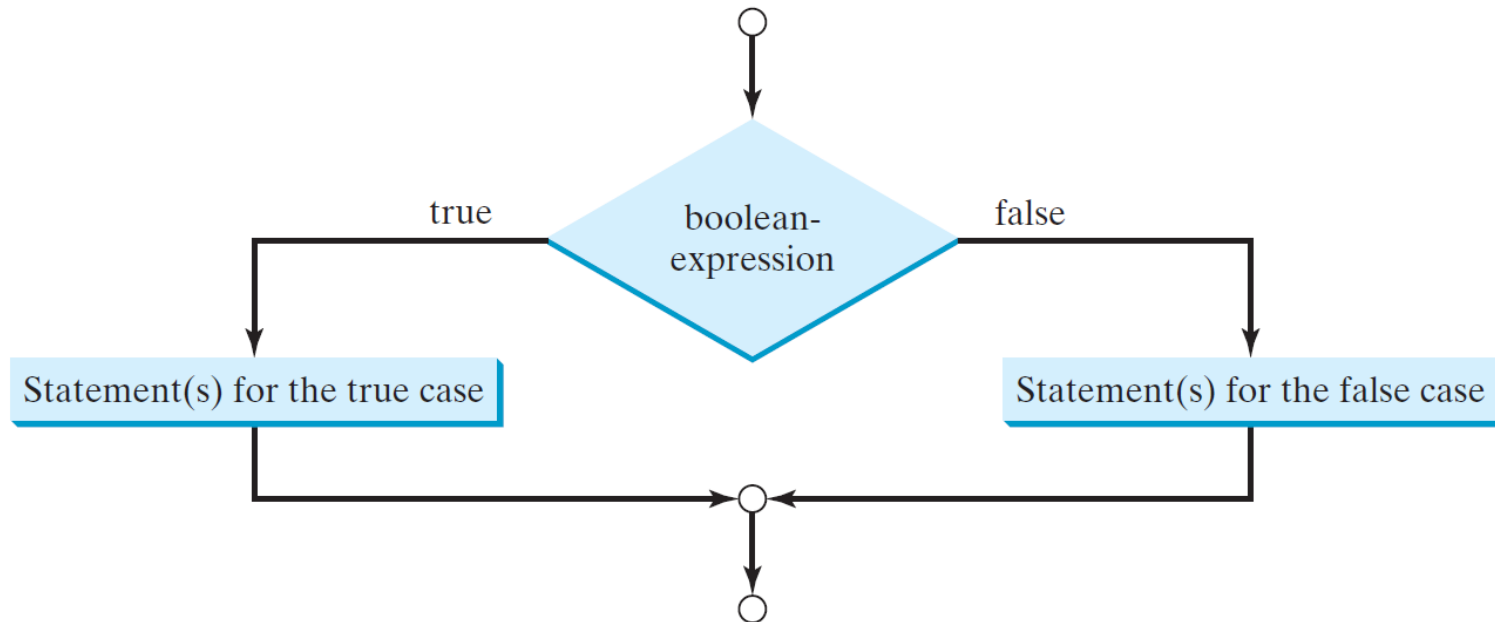
```
if (radius >= 0) {  
    area = radius * radius * PI;  
    System.out.println("The area for  
the circle of radius "  
        + radius + " is " + area);  
}
```



# if-else statements

```
if (boolean-expression) {  
    statement(s)-for-the-true-case;  
}  
else {  
    statement(s)-for-the-false-case;  
}
```

Braces are optional for a single statement; however, it is best practice (less error prone) to **always use braces**



# if-else statements

```
if (radius >= 0) {  
    area = radius * radius * 3.14159;  
    System.out.println("The area for the "  
        + "circle of radius " + radius +  
        " is " + area);  
}  
else {  
    System.out.println("Negative input");  
}
```

# Conditional operator

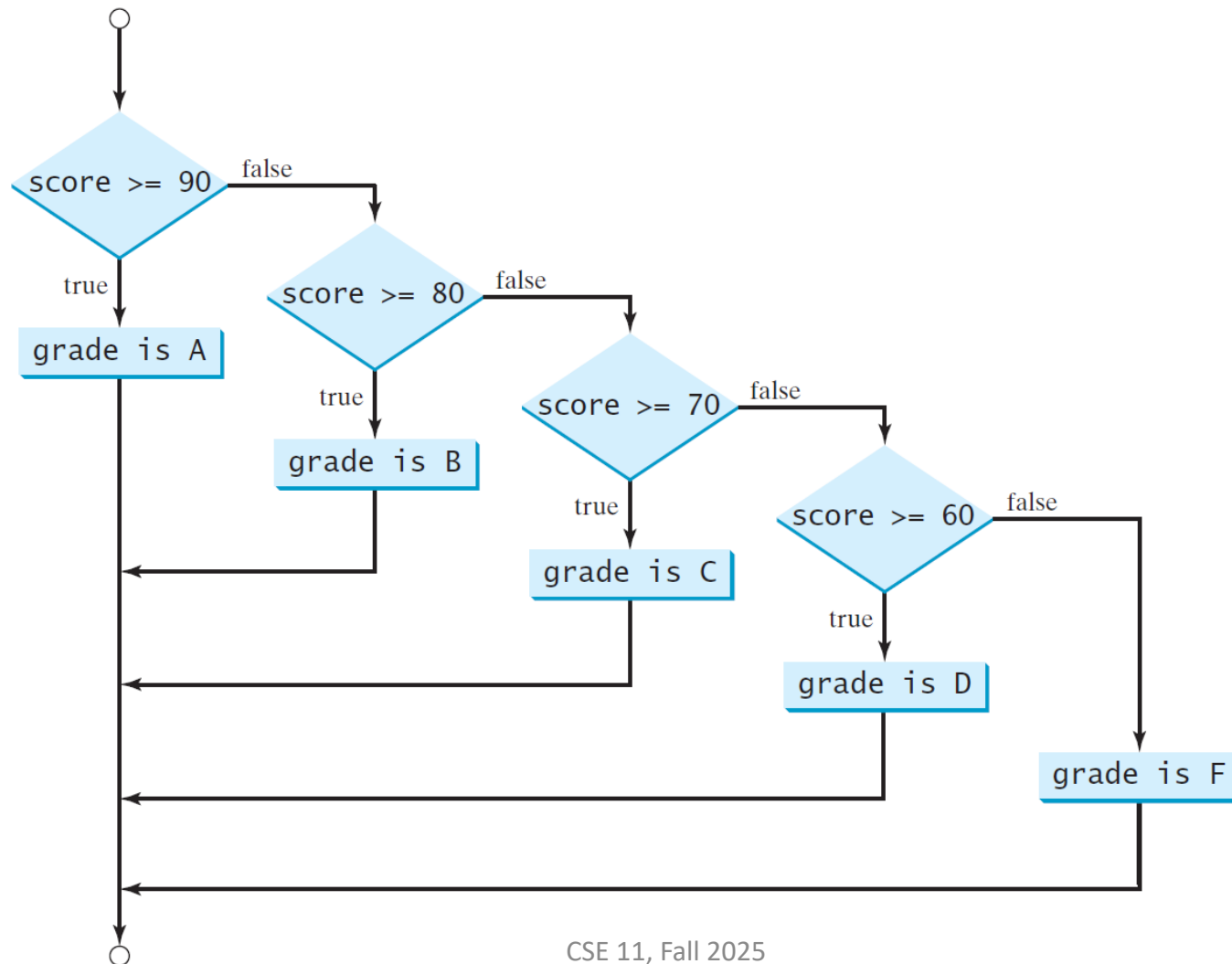
`(boolean-expression) ? expression1 : expression2`

```
if (x > 0) {  
    y = 1;  
}  
else {  
    y = -1;  
}
```

is equivalent to

```
y = (x > 0) ? 1 : -1;
```

# Multiple if-else statements



# Multiple if-else statements

```
if (score >= 90.0)
    System.out.print("A");
else
    if (score >= 80.0)
        System.out.print("B");
    else
        if (score >= 70.0)
            System.out.print("C");
        else
            if (score >= 60.0)
                System.out.print("D");
            else
                System.out.print("F");
```

(a)

Equivalent

```
if (score >= 90.0)
    System.out.print("A");
else if (score >= 80.0)
    System.out.print("B");
else if (score >= 70.0)
    System.out.print("C");
else if (score >= 60.0)
    System.out.print("D");
else
    System.out.print("F");
```

This is better

(b)

# Nested statements

- The `else` clause matches the **most recent if** clause in the same block

```
int i = 1, j = 2, k = 3;

if (i > j)
  if (i > k)
    System.out.println("A");
  else
    System.out.println("B");
```

(a)

Equivalent

This is better  
with correct  
indentation

```
int i = 1, j = 2, k = 3;

if (i > j)
  if (i > k)
    System.out.println("A");
  else
    System.out.println("B");
```

(b)

Braces are optional for a single statement; however, it is best practice (less error prone) to **always use braces**

## Nothing is printed

# Nested statements

- To force the `else` clause to match the first `if` clause, you must add a pair of braces

```
int i = 1;
int j = 2;
int k = 3;
if (i > j) {
    if (i > k)
        System.out.println("A");
}
else
    System.out.println("B");
```

Braces are optional for a single statement; however, it is best practice (less error prone) to **always use braces**

**B is printed**

# Less error prone

```
if (number % 2 == 0)
    even = true;
else
    even = false;
```

(a)

Equivalent

```
boolean even
    = number % 2 == 0;
```

(b)

These are better

```
if (even == true)
    System.out.println(
        "It is even.");
```

(a)

Equivalent

```
if (even)
    System.out.println(
        "It is even.");
```

(b)

# Logical operators

Operator	Name	Description
!	not	logical negation
&&	and	logical conjunction
	or	logical disjunction
^	exclusive or (xor)	logical exclusion

# Truth table for operator !

p	!p	Example: age = 24 and weight = 140
true	false	!(age > 18) is false, because (age > 18) is true
false	true	!(weight == 150) is true, because (weight == 150) is false

# Truth table for operator &&

$p_1$	$p_2$	$p_1 \ \&\& \ p_2$	Example: age = 24 and weight = 140
false	false	false	<code>(age &lt;= 18) &amp;&amp; (weight &lt; 140)</code> is false, because both conditions are false
false	true	false	<code>(age &lt;= 18) &amp;&amp; (weight &gt;= 140)</code> is false, because <code>(age &lt;= 18)</code> is false
true	false	false	<code>(age &gt; 18) &amp;&amp; (weight &gt; 140)</code> is false, because <code>(weight &gt; 140)</code> is false
true	true	true	<code>(age &gt; 18) &amp;&amp; (weight &gt;= 140)</code> is true, because both conditions are true

# Truth table for operator `||`

$p_1$	$p_2$	$p_1    p_2$	Example: age = 24 and weight = 140
false	false	false	<code>(age &gt; 34)    (weight &gt;= 150)</code> is false, because both conditions are false
false	true	true	<code>(age &gt; 34)    (weight &lt;= 140)</code> is true, because <code>(weight &lt;= 140)</code> is true
true	false	true	<code>(age &gt; 14)    (weight &gt;= 150)</code> is false, because <code>(age &gt; 14)</code> is true
true	true	true	<code>(age &gt; 14)    (weight &lt;= 140)</code> is true, because both conditions are true

# Truth table for operator $\wedge$

$p_1$	$p_2$	$p_1 \wedge p_2$	Example: age = 24 and weight = 140
false	false	false	$(\text{age} > 34) \wedge (\text{weight} > 140)$ is false, because both conditions are false
false	true	true	$(\text{age} > 34) \wedge (\text{weight} \geq 140)$ is true, because $(\text{age} > 34)$ is false and $(\text{weight} \geq 140)$ is true
true	false	true	$(\text{age} > 14) \wedge (\text{weight} > 140)$ is true, because $(\text{age} > 14)$ is true and $(\text{weight} > 140)$ is false
true	true	false	$(\text{age} > 14) \wedge (\text{weight} \geq 140)$ is false, because both conditions are true

# Short-circuit operators

- `&&` and `||` are short-circuit operators
- `p1 && p2`
  - If `p1` or `p2` is false, then `p1 && p2` is false
  - `p1` is evaluated first
    - If `p1` is true, then `p2` is evaluated
    - If `p1` is false, then `p2` is **not** evaluated
- `p1 || p2`
  - If `p1` or `p2` is true, then `p1 || p2` is true
  - `p1` is evaluated first
    - If `p1` is true, then `p2` is **not** evaluated
    - If `p1` is false, then `p2` is evaluated

# switch statements

- When the value in a case statement matches the value of the switch-expression, the statements starting from this case are executed until either a break statement or the end of the switch statement is reached

```
switch (switch-expression) {  
    case value1:  statement(s)1;  
                break;  
    case value2: statement(s)2;  
                break;  
    ...  
    case valueN: statement(s)N;  
                break;  
    default:    statement(s)-for-default;  
}
```

# switch statements

- The switch-expression must yield a value of char, byte, short, int or String type and must always be enclosed in parentheses
- The value1, ..., and valueN must have the *same data type* as the value of the switch-expression
- The resulting statements in the case statement are executed when the value in the case statement matches the value of the switch-expression
- Note that value1, ..., and valueN are *constant expressions* (i.e., they cannot contain variables in the expression, such as `1 + x`)

```
switch (switch-expression) {  
    case value1: statement(s)1;  
                break;  
    case value2: statement(s)2;  
                break;  
    ...  
    case valueN: statement(s)N;  
                break;  
    default: statement(s)-for-default;  
}
```

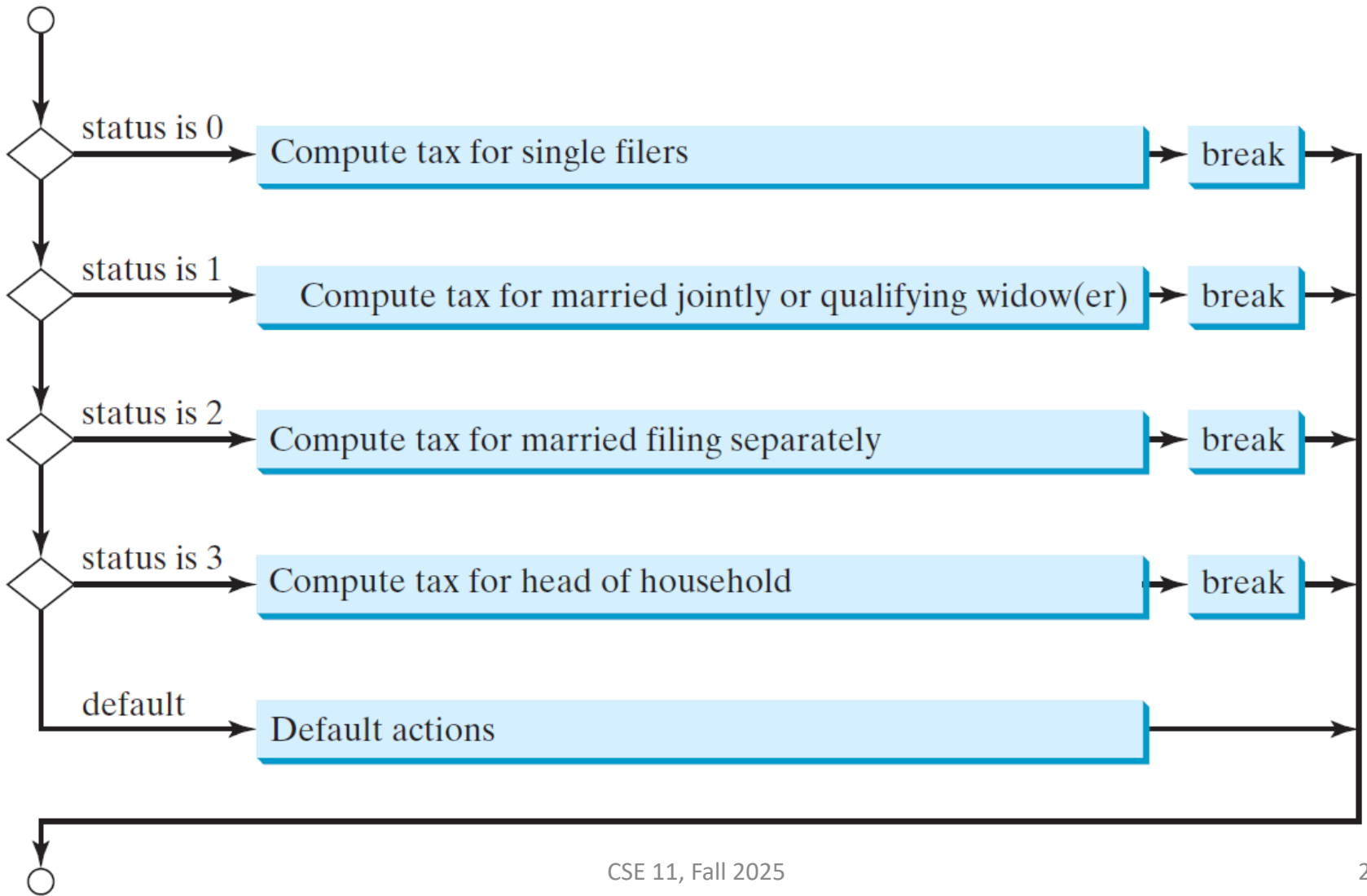
# switch statements

- The keyword `break` is optional, but it should be used at the end of each case in order to terminate the remainder of the `switch` statement
  - If the `break` statement is not present, the next case statement will be executed
- The `default` case, which is optional, can be used to perform actions when none of the specified cases matches the `switch-expression`

```
switch (switch-expression) {  
    case value1: statement(s)1;  
                break;  
    case value2: statement(s)2;  
                break;  
    ...  
    case valueN: statement(s)N;  
                break;  
    default: statement(s)-for-default;  
}
```

The `default` case is optional; however, it is best practice (less error prone) to **always have a default case**

# switch statements



# switch statements

```
switch (status) {  
    case 0:  compute taxes for single filers;  
             break;  
    case 1:  compute taxes for married file jointly;  
             break;  
    case 2:  compute taxes for married file separately;  
             break;  
    case 3:  compute taxes for head of household;  
             break;  
    default: System.out.println("Error: invalid status");  
             System.exit(1);  
}
```

The default case is optional;  
however, it is best practice (less error  
prone) to **always have a default case**

# switch statements

```
switch (day) {  
    case 1:  
    case 2:                                Fall-through  
    case 3:  
    case 4:  
    case 5:  
System.out.println("Weekday");  
break;  
    case 0:                                Fall-through  
    case 6:  
System.out.println("Weekend");  
}
```

# Operator precedence

- `()`, `var++`, `var--`
- `++var`, `--var`, `+`, `-` (unary plus and minus), `!` (not)
- (type) casting
- `*`, `/`, `%` (multiplication, division, and remainder)
- `+`, `-` (binary addition and subtraction)
- `<`, `<=`, `>`, `>=` (relational operators)
- `==`, `!=` (equality)
- `^` (exclusive or)
- `&&` (and)
- `||` (or)
- `=`, `+=`, `-=`, `*=`, `/=`, `%=` (assignment operators)

# Operator associativity

- When two operators with the same precedence are evaluated, the associativity of the operators determines the order of evaluation
- All binary operators except assignment operators are left-associative
  - $a - b + c - d$  is equivalent to  $((a - b) + c) - d$
- Assignment operators are right-associative
  - $a = b += c = 5$  is equivalent to  $a = (b += (c = 5))$

# Operator precedence and associativity

- The expression in the parentheses is evaluated first
  - Parentheses can be nested, in which case the expression in the inner parentheses is executed first
- When evaluating an expression without parentheses, the operators are applied according to the precedence rule and the associativity rule
- If operators with the same precedence are next to each other, their associativity determines the order of evaluation

# Next Lecture

- Methods